Scientific Report for the Mandelbrot Visualization Project

Maurice Amon[1]

May 22, 2025

Department of Informatics - M. Sc. Course in Concurrency and Multicore Programming

Département d'Informatique - Departement für Informatik • Université de Neuchatel - Universität Neuchatel • Switzerland

[1] maurice.amon@students.unibe.ch, Student @ Swiss Joint Master in Computer Science

Contents

1 Technical Report

The following report gives a short overview over how I've realized the semester project in the course "Concurrency: Multicore Programming and Data Processing" at the University of Neuchatel.

1.1 Programming Language & Frameworks

The program has been implemented in Java but instead of the classical Swing Framework, I've made use of JavaFX for the GUI as I think it offers a more modern design.

1.2 Mathematical & Engineering foundations

1.2.1 Construction of the Mandelbrot set

The Mandelbrot set is a mathematical set within the complex numbers that has been defined by Benoit Mandelbrot. For an arbitrary complex number c, it defines the following recursively defined function:

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

The complex number c belongs to the Mandelbrot set M if it satisfies the following condition:

$$M = \left\{ c \in C \;\middle|\; \lim_{n\to\infty} |z_n| \leq 2 \right\}$$

As a programmer, this is difficult to check, as we'd have to iterate through the function infinitely often. However, it is enough to approximate it by choosing a sufficiently high n. In the project, the default value of n is set to n = 1000, in case the user does not define it by himself.

1.2.2 Mapping of the Canvas Pixels to a complex number

As the domain of complex numbers that we are going to consider is in [-2.5; 1.0] for the real part and [-1.5; 1.5] for the imaginary part, we cannot directly use the coordinates of the Canvas pixels, since it is of size 1200 width times 700 pixels in height. Therefore, a mapping is required that takes a pixel coordinate as an input und outputs a complex number within our domain. The function is defined as follows:

$$f : \{0, 1, \ldots, W-1\} \times \{0, 1, \ldots, H-1\} \to C$$

Whereas W is the width of our Canvas and H the height. We substract one as we start with the pixel index 0. $x_{\min}$ represents the lower bound of the range of the complex numbers that we are going to consider for the Mandelbrot set construction, while $x_{\max}$ corresponds to the upper bound.

$$f(p_x, p_y) = \left(x_{\min} + \frac{p_x}{W-1} \cdot (x_{\max} - x_{\min}) \right) + i \left(y_{\max} - \frac{p_y}{H-1} \cdot (y_{\max} - y_{\min}) \right)$$

1.2.3 Mapping of the iteration count to a RGB color

The iteration count n defines after which iteration we could decide whether c belongs to the Mandelbrot set M or not.

As a first step we've the absolute value of the complex number z_n:

$$|z_n| = \sqrt{a^2 + b^2}$$

By applying the smooth iteration count we get a value that we can use for smooth coloring: [2]

$$f(n, z) = (n+1) - \frac{\log(\log(|z_n|))}{\log(2)}$$

I'm then using the result of this equation and normalize it s. t. we obtain a value within the range of [0, 1] for a sufficiently large $Max_iterations$. As the RGB-Color space is used in this project we can then multiply the result by 255, floor the double value s.t. we obtain an integer, to obtain a corresponding RGB-value t ∈ [0, 255], as defined by the following formula:

$$t = \left\lfloor 255 \cdot \sqrt{\frac{\mathrm{f(n,\ z)} - \mathrm{Min_{iteration}}}{\mathrm{Max_{iteration}} - \mathrm{Min_{iteration}}}} \right\rfloor$$

However, for a relatively small $\mathrm{Max_{iteration}}$ the obtained value can exceed 255 - so we've to check and if it is the case, we map the value to 255.

The variable $t \in [0, 255]$ is then used to create a RGB value (0, 0, t) for the corresponding pixel of the complex number.

1.2.4 Multithreading

As we can independently decide for each complex number whether it belongs to the Mandelbrot set or not, we can split the work for doing so among different threads, to decrease the overall runtime. As I'm using JavaFX for the display of the Mandelbrot visualization I'd to consider that updating the UI after each calculated pixel results in a high runtime as we'd have to switch the threads after each calculated pixel. I therefore decided to calculate the whole Mandelbrot set first and then update the UI. Depending on the amount of threads N the user chooses, the 2-dimensional pixel array of the canvas is vertically splitted in N sub-arrays from which each thread is going to process one. As we've to wait until all threads have finished their work, before we are going to update the UI, I've decided to use a CountDownLatch from the package java.util.concurrent.

After all these considerations I still had suboptimal runtimes. As the program calculates the mandelbrot set rather fast (a matter of seconds) I figured out that the overhead I experienced was most likely due to the fact that the creation of threads needs a few milliseconds too. To solve this I instantiated a Executor Service for a Threadpool of 10 threads at the start of the application. The threads are then dynamically used during the runtime of the program.

1.2.5 How to use

You can download and compile the program in Intellij and then start it. As mentioned before, the UI allows the user in a Toolbar to set the amount of threads as well as the iterations that the Mandelbrot algorithm is going to use. Be aware, that only values between 1 and 10 are considered als valid inputs for the amount of threads. If your input is outside of that range it will be automatically be set to 1. Also keep in mind, that negative values for the iteration-field are not possible, it will be automatically reset to 1000 if you declare it. The repository is publicly available on my Github page. [1]

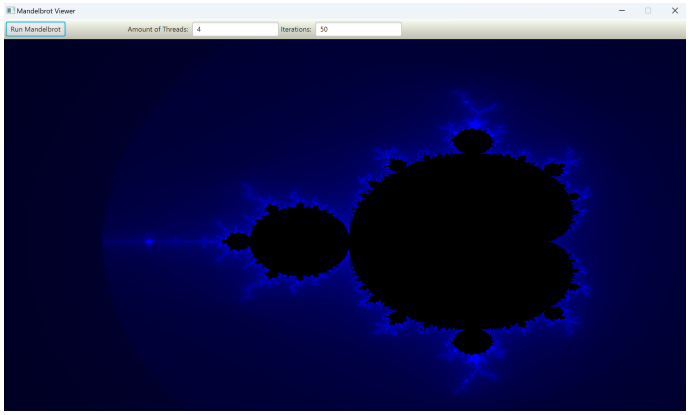

Figure 1: Mandelbrot Visualizer by Maurice Amon

2 Evaluation

The evaluation has been conducted with two thread configurations, as the project description explicitly stated to compare the single-threaded version to a configuration using 4 threads.

2.1 Hardware

The tests have been conducted locally with a Dell Precision 5570 Laptop with the following Hardware specifications:

- CPU: Intel® Core™ i9-12900H
- RAM: 32 GB DDR5
- GPU: Nvida RTX A2000

2.2 Test

Amound of Threads	**First trial: Runtime**	**Second trial: Runtime**	**Third trial: Runtime**
1 Thread	1938 ms	1558 ms	1453 ms
4 Threads	787 ms s	727 ms	694 ms

Table 1: Runtime comparison of single-threaded vs. multi-threaded.

This implies that we've on average a speedup-factor of $\frac{(1938+1558+1453)ms}{(787+727+694)ms} = 2.241$, which satisfies the condition stated in the project description.

References

[1] Maurice Amon. Mandelbrot visualizer. https://github.com/Konpyuuta/Mandelbrot-Visualizer, 2025. Accessed May 2025.

[2] Balise's Blog. Smooth coloring of mandelbrot. https://blogen.pasithee.fr/2019/01/06/smooth-coloring-of-mandelbrot/, 2019. Accessed May 2025.